A Distributed Abstraction Algorithm for Online Predicate Detection

Himanshu Chauhan   Vijay K. Garg
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
himanshu@utexas.edu   garg@ece.utexas.edu

Aravind Natarajan   Neeraj Mittal
Dept. of Computer Science
The University of Texas at Dallas
aravindn@utdallas.edu   neerajm@utdallas.edu

***Abstract*—Analyzing a distributed computation is a hard problem in general due to the combinatorial explosion in the size of the state-space with the number of processes in the system. By abstracting the computation, unnecessary state explorations can be avoided. Computation slicing is an approach for abstracting distributed computations with respect to a given predicate. We focus on regular predicates, a family of predicates that covers many commonly used predicates for runtime verification. The existing algorithms for computation slicing are centralized – a single process is responsible for computing the slice in either offline or online manner. In this paper, we present first distributed online algorithm for computing the slice of a distributed computation with respect to a regular predicate. Our algorithm distributes the work and storage requirements across the system, thus reducing the space and computation complexity per process.**

I. Introduction

Global predicate detection [1] for runtime verification is an important technique for detecting violations of invariants for debugging and fault-tolerance in distributed systems. It is a challenging task on a large system with a large number of processes due to the combinatorial explosion of the state space. The predicate detection problem is not only applicable to conventional distributed systems, but also to multicore computing. With growing popularity of large number of CPU-cores on processing chips [2], some manufacturers are exploring the distributed computing model on chip with no shared memory between the cores, and information exchange between the cores only using message passing [3]. Recent research efforts [4] have shown that with sufficiently fast on-chip networking support, such a message passing based model can provide significantly fast performance for some specific computational tasks. With emergence of these trends, techniques in predicate detection for distributed systems can also be useful for systems with large number of cores.

Multiple algorithms have been proposed in literature for detection of global predicates in both offline and online manner (e.g. [1], [5], [6]). Online predicate detection is important for many system models: continuous services (such as webservers), collection of continuous observations (such as sensor-networks), and parallel search operations on large clusters. However, performing predicate detection in a manner that is oblivious to the structure of the predicate can lead to large runtime, and high memory overhead. The approach of using mathematical abstractions for designing and analyzing computational tasks has proved to be significantly advantageous in modern computing. In the context of predicate detection, and runtime verification, the problem of abstraction can be viewed as the problem of taking a distributed computation as input and outputting a smaller distributed computation that abstracts out parts that are not relevant to the predicate under consideration. The abstract computation may be exponentially smaller than the original computation resulting in significant savings in predicate detection time.

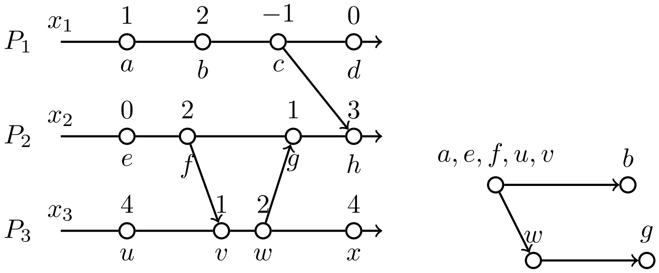
Computation slicing is an abstraction technique for efficiently finding all global states, of a distributed computation, that satisfy a given global predicate, without explicitly enumerating all such global states [5]. The *slice* of a computation with respect to a predicate is a sub-computation that satisfies the following properties: (a) it contains all global states of the computation for which the predicate evaluates to true, and (b) of all the sub-computations that satisfy condition (a), it has the least number of global states. As an illustration, consider the computation shown in Fig. 1(a). The computation consists of three processes P_1, P_2, and P_3 hosting integer variables x_1, x_2, and x_3, respectively. An event, represented by a circle is labeled with the value of the variable immediately after the event is executed.

Suppose we want to determine whether the property (or the predicate) $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geq 1) \wedge (x_3 \leq 3)$ ever holds in the computation. In other words, does there exist a global state of the computation that satisfies the predicate? The predicate could represent the violation of an invariant. Without computation slicing, we are forced to examine all global states of the computation, twenty-eight in total, to ascertain whether some global state satisfies the predicate. Alternatively, we can compute a slice of the computation automatically with respect to the predicate $(x_1 \geq 1) \wedge (x_3 \leq 3)$ as shown in Fig. 1(b).

We can now restrict our search to the global states of the slice, which are only six in number, namely: $\{a, e, f, u, v\}$, $\{a, e, f, u, v, b\}$, $\{a, e, f, u, v, w\}$, $\{a, e, f, u, v, b, w\}$, $\{a, e, f, u, v, w, g\}$, and $\{a, e, f, u, v, b, w, g\}$.

The slice has much fewer global states than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

In this paper, we focus on abstracting distributed computations with respect to *regular* predicates (defined in Sec. II). The



(a) Computation

(b) Slice

Fig. 1: A Computation, and its slice with respect to predicate $(x_1 \geq 1) \wedge (x_2 \leq 3)$

family of *regular* predicates contains many useful predicates that are often used for runtime verification in distributed systems. Some such predicates are:

Conjunctive Predicates: Global predicates which are conjunctions of local predicates. For example, predicates of the form, $B = (l_1 \geq x_1 \geq u_1) \wedge (l_2 \geq x_2 \geq u_2) \wedge \ldots \wedge (l_n \geq x_n \geq u_n)$, where x_i is the local variable on process P_i, and l_i, u_i are constants, are conjunctive predicates. Some useful verification predicates that are in conjunctive form are: detecting mutual exclusion violation in pairwise manner, pairwise data-race detection, detecting if each process has executed some instruction, etc.

Monotonic Channel Predicates [7]: Some examples are: all messages have been delivered (or all channels are empty), at least k messages have been sent/received, there are at most k messages in transit between two processes, the leader has sent all “prepare to commit” messages, etc.

Centralized offline [8] and online [9] algorithms for *slicing* based predicate detection have been presented previously. In this paper, we present the first *distributed online* slicing algorithm for *regular* predicates in distributed systems.

A. Challenges and Contributions

Computing the slice of a computation with respect to a general predicate is a NP-Hard problem in general [8]. Many classes of predicates have been identified for which the slice can be computed efficiently in polynomial time (*e.g.*, regular predicates, co-regular predicates, linear predicates, relational predicates, stable predicates) [8], [5], [9], [10]. However, the existing slicing algorithms are *centralized* in nature. The slice is computed by a single *slicer* process that examines every relevant event in the computation. The centralized algorithms may be *offline*, where all events are known a priori, or *online*, where the slice is updated incrementally with the arrival of every new relevant event. For systems with large number of processes, such centralized algorithms require a single process to perform high number of computations, and to store very large data. In comparison, a distributed online algorithm significantly reduces the per process costs for both computation and storage. Distributed algorithms are generally faster in comparison to centralized algorithms and allow division of tasks among multiple processes. Additionally, for predicate detection, the centralized online algorithm requires at least one message to the slicer process for every relevant event in the computation, resulting in a bottleneck at the slicer process.

A method of devising a distributed algorithm from a centralized algorithm is to decompose the centralized execution steps into multiple steps to be executed by each process independently. However, for performing online abstraction using computation slicing, such an incremental modification would lead to a large number of messages, computational steps, and memory overhead. The reason for this inefficiency is that by directly decomposing the steps of the centralized online algorithm, the *slicing* computation would require each process to send its local state information to all the other processes whenever the local state (or state interval) is updated. In addition, only splitting the centralized algorithm across all processes leads to a distributed algorithm that wastes significant computational time as multiple processes may end up visiting (and enumerating) the same global state. Thus, the task of devising an efficient distributed algorithm for *slicing* is non-trivial. In this paper, we propose a distributed algorithm that exploits not only the nature of the predicates, but also the collective knowledge across processes. The optimized version of our algorithm reduces the required storage per *slicing* process, and computational workload per *slicing* process by $O(n)$. An experimental evaluation of our proposed approach with the centralized approach can be found in the extended version of this paper [11].

B. Applications

Our algorithm is useful for global predicate detection. Suppose the predicate B is of the form $B_1 \wedge B_2$, where B_1 is regular but B_2 is not. We can use our algorithm to slice with respect to B_1 to reduce the time and space complexity of the global predicate detection. Instead of searching for the global state that satisfies B in the original computation, with the distributed algorithm we can search the global states in the slice for B_1. For example, the Cooper-Marzullo algorithm traverses the lattice of global states in an online manner [1], which can be quite expensive. By running our algorithm together with Cooper-Marzullo algorithm, the space and time complexity of predicate detection is reduced significantly (possibly exponentially) for predicates in the above mentioned form.

Our algorithm is also useful for recovery of distributed programs based on checkpointing. For fault-tolerance, we may want to restore a distributed computation to a checkpoint which satisfies the required properties such as “all channels are empty”, and “all processes are in some states that have been saved on storage”. If we compute the slice of the computation in an online fashion, then on a fault, processes can restore the global state that corresponds to the maximum of the last vector of the slice at each surviving process. This global state is consistent as well as recoverable from the storage.

II. Background: Regular Predicates and Slicing

A. Model

We assume a loosely coupled asynchronous message passing system, consisting of n reliable processes (that do not fail), denoted by $\{P_1, P_2, \ldots, P_n\}$, without any shared memory or global clock. Channels are assumed to be FIFO, and lossless. In our model, each local state change is considered an event; and every message activity (send or receive) is also represented by a new event. We assume that the computation being analyzed does not deadlock.

A distributed computation is modeled as a partial order on a set of events [12], given by Lamport's *happened-before* ($\rightarrow$) relation [12]. We use $(E, \rightarrow)$ to denote the distributed computation on a set of events E. Mattern [13] and Fidge [14] proposed *vector clocks*, an approach for time-stamping events in a computation such that the happened-before relation can be tracked. If V denotes the vector clock for an event e in a distributed computation, then for any event f in the computation: $e \rightarrow f \Leftrightarrow e.V < f.V$. For any pair of events e and f such that $e \not\rightarrow f \wedge f \not\rightarrow e$, e and f are said to be concurrent, and this relation is denoted by $e||f$. Fig. 2(a) shows a sample distributed computation, and its corresponding vector clock representation is presented in Fig. 2(b).

We now present some required concepts:

Definition 1 (Consistent Cut)**.** *Given a distributed computation $(E, \rightarrow)$, a subset of events $C \subseteq E$ is said to form a* consistent cut *if C contains an event e only if it contains all events that happened-before e. Formally, $e \in C \wedge f \rightarrow e \implies f \in C$.*

The concept of a consistent cut (or, a consistent global state) is identical to that of a down-set (or order-ideal) used in lattice theory [15]. Intuitively, a consistent cut captures the notion of a global state of the system at some point during its execution [16].

Consider the computation shown in Fig 2(a). The subset of events $\{a, b, e\}$ forms a consistent cut, whereas the subset $\{a, e, f\}$ does not; because $b \rightarrow f$ (b happened-before f) but b is not included in the subset. A consistent cut can also be represented with a vector clock notation. For any consistent cut C, its vector clock $C.V$ can be computed as $C.V =$ component-wise-max$\{e.V \mid event\ e \in C\}$, where $e.V$ denotes the vector clock of event e. For this paper, we use a shortened notation for a cut of the computation. A cut C is denoted by the latest events on each process. Thus, $\{a, b, e\}$ is denoted by $[b, e]$ and $\{a, e, f\}$ is represented as $[a, f]$.

Table I shows all the consistent global states/cuts and their corresponding vector clock values for the computation in Fig. 2. We now present additional notions from lattice theory that are key to our approach.

Definition 2 (Join)**.** *A join of a pair of global states is defined as the set-union of the set of events in the states.*

Definition 3 (Meet)**.** *A meet of a pair of global states is defined as the set-intersection of the set of events in the states.*

For two global states C_1 and C_2, their join is denoted with $C_1 \sqcup C_2$, whereas $C_1 \sqcap C_2$ denotes their meet.

Theorem 1. *[15], [13] Let $\mathcal{C}(E)$ denote the set of all consistent cuts of a computation $(E, \rightarrow)$. $\mathcal{C}(E)$ forms a lattice under the relation $\subseteq$.*

A *global predicate* (or simply a *predicate*) is a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated on the state resulting after executing all events in the cut. A global predicate is *local* if it depends only on variables of a single process. If a predicate B evaluates to true for a consistent cut C, we say that "C satisfies B" and denote it by C_B.

Definition 4 (Linearity Property of Predicates)**.** *A predicate B is said to have the linearity property, if for any consistent cut C, which does not satisfy predicate B, there exists a process P_i such that a cut that satisfies B can never be reached from C without advancing along P_i.*
Predicates that have the linearity property are called **linear predicates***.*

For example, consider the cut $[b, e]$ of the computation shown in Fig. 2(a). The cut does not satisfy the predicate "all channels are empty", and for the given cut, progress must be made on P_2 to reach the cut $[b, f]$ which satisfies the predicate.

The process P_i in the above definition is called a *forbidden process*. For a computation involving n processes, given a consistent cut that does not satisfy the predicate, P_i can be found in $O(n)$ time for most linear predicates used in practice. To find a *forbidden process* given a consistent cut, a process first checks if the cuts needs to be advanced on itself; if not it checks the states in the total order defined using process ids, and picks the first process whose state makes the predicate false on the cut. The set of linear predicates has a subset, the set of *regular predicates*, that exhibits a stronger property.

Definition 5 (Regular Predicates)**.** *A predicate is called* regular *if for any two consistent cuts C and D that satisfy the predicate, the consistent cuts given by $(C \sqcap D)$ (the meet of C and D) and $(C \sqcup D)$ (the join of C and D) also satisfy the predicate.*

TABLE I: Consistent Global States of Fig. 2 and Predicate Evaluation for B="*all channels empty*"

#	State	Cut Vec. Clock	Pred. Eval.
1	[]	$[0,0]$	**True**
2	[a]	$[1,0]$	**True**
3	[b]	$[2,0]$	False
4	[c]	$[3,0]$	False
5	[e]	$[0,1]$	**True**
6	[a,e]	$[1,1]$	**True**
7	[b,e]	$[2,1]$	False
8	[b,f]	$[2,2]$	**True**
9	[b,g]	$[2,3]$	**True**
10	[c,e]	$[3,1]$	False
11	[c,f]	$[3,2]$	**True**
12	[c,g]	$[3,3]$	**True**

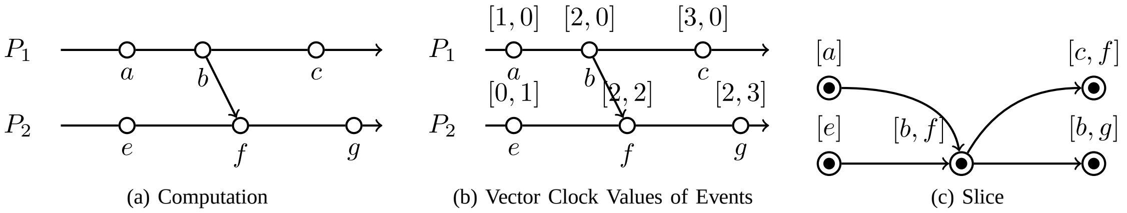


Fig. 2: A Computation, Vector Clock Representation, and Slice with respect to predicate $B =$“all channels are empty”

Examples of regular predicates include local predicates (*e.g.*, $x \leq 4$), conjunction of local predicates (*e.g.*, $(x \leq 4) \wedge (y \geq 2)$ where x and y are variables on different processes) and monotonic channel predicates (*e.g.*, there are at most k messages in transit from P_i to P_j) [8]. Table I indicates whether or not the predicate “all channels empty” is satisfied by each of the consistent global cuts of the computation in Fig. 2. To use *computation slicing* for detecting regular predicates, we first need to capture the notion of *join-irreducible elements* for the lattice of consistent cuts.

Definition 6 (Join-Irreducible Element). *Let L be a lattice. An element $x \in L$ is join-irreducible if*

1) *x is not the smallest element of L*
2) *$\forall a, b \in L : (x = a \sqcup b) \implies (x = a) \vee (x = b)$.*

Intuitively, a join-irreducible element of a lattice is one that cannot be represented as the join of two distinct elements of the lattice, both different from itself. For the lattice of consistent cuts of a distributed computation, the join-irreducible elements correspond to consistent cuts that can not be reached by joins (set-union) of two or more consistent cuts. For the computation of Fig. 2, the join-irreducible consistent cuts are: $[a], [b], [c], [e], [b, f], [b, g]$.

B. Computation Slice

A computation slice of a computation with respect to a predicate B is a concise representation of all the consistent cuts of the computation that satisfy the predicate B. When the predicate B is regular, the set of consistent cuts satisfying B, L_B forms a sublattice of L, that is the lattice of all consistent cuts of the computation $(E, \rightarrow)$. L_B can equivalently be represented using its join-irreducible elements [15]. Intuitively, join-irreducible elements form the basis of the lattice. The lattice can be generated by taking joins of its basis elements. Let J_B be the set of all join-irreducible elements of L_B. Let $J_B(e)$ denote the least consistent cut that includes e and satisfies B. Then, it can be shown [17] that

$$J_B = \{J_B(e) | e \in E\}$$

The $J_B(event)$ values, in vector clock notation, for each event of the computation in Fig. 2 are:
$J_B(a) = [1, 0]$, $J_B(b) = [2, 2]$, $J_B(c) = [3, 2]$, $J_B(e) = [0, 1]$, $J_B(f) = [2, 2]$, $J_B(g) = [2, 3]$. We can now define a computation slice formally.

Definition 7. *Let $(E, \rightarrow)$ be a computation and B be a regular predicate. The slice of the computation with respect to B is defined as $(J_B, \subseteq)$.*

The definition given here is different from the one given in [17] and [8] but equivalent for regular predicates as shown in [8].

Note: It is important to observe that $J_B(e)$ does not necessarily exist. Also, multiple events may have the same $J_B(e)$.

For the computation shown in Fig. 2(a), Fig. 2(c) presents a visual representation of the slice.

A centralized online algorithm to compute J_B was proposed in [5]. In the online version of this centralized algorithm, a pre-identified process, called *slicer* process, plays the role of the slice computing process. All the processes in the system send their event and local state values whenever their local states change. The *slicer* process maintains a queue of events for each process in the system, and on receiving the data from a process adds the event to the relevant queue. In addition, the slicer process also keeps a map of events and corresponding local states for each process in the system. For each received event, the slicer appends the event and local state mapping to the respective map. For every event e it receives, the slicer computes $J_B(e)$ using the *linearity property*.

The centralized approach suffers from the drawback of causing a heavy load of messages as well as computation on just one process, namely the *slicer* process. Thus, for any large distributed computation, this approach would not scale well. To address this issue, we propose a distributed algorithm that significantly reduces the computational, as well as the message load on any process.

III. A Distributed Online algorithm for Slicing

In this section, we present the key ideas and routines for distributed online algorithm for computing the slice. The required optimizations that tackle the challenges listed in Section I-A are discussed later. In our algorithm, we have n slicer processes, $S_1, S_2, ..., S_n$, one for every application process. All slicer processes cooperate to compute the task of slicing $(E, \rightarrow)$. Let E be partitioned into n sets E_i such that E_i is the set of events that occurred in P_i. In our algorithm, S_i computes

$$J_i(B) = \{J_B(e) | e \in E_i\}$$

Observe that by the definition of join-irreducible consistent cut, $e \rightarrow f$ implies $J_B(e) \subseteq J_B(f)$. Since all events in a process are totally ordered, the set of consistent cuts generated by any S_i are also totally ordered. **Note**: In this paper, the symbol → indicates a *happened-before* relation; whereas the symbol ← in the pseudo-code denotes assignment operation.

Algorithm 1 presents the distributed algorithm for online slicing with respect to a regular predicate B. Each slicer process has a token assigned to it that goes around in the system. Other slicer processes cooperate in maintaining and processing the token. The goal of the token for the slicer process S_i is to compute $J_B(e)$ for all events $e \in E_i$. Whenever the token has computed $J_B(e)$ it returns to its original process, reports $J_B(e)$ and starts computing $J_B(succ(e))$, $succ(e)$ being the immediate successor of event e. The token T_i carries with it the following data:

- pid: Process id of the slicer process to which it belongs.
- $event$: Details of event e, specifically the event id and event's vector clock, at P_i for which this token is computing $J_B(e)$. The identifier for event e is the tuple $<pid, eid>$ that identifies each event in the computation uniquely.
- $gcut$: The vector clock corresponding to the cut which is under consideration (a candidate for $J_B(e)$).
- $depend$: Dependency vector for events in $gcut$. The dependency vector is updated each time the information of an event is added to the token (steps explained later), and is used to decide whether or not some cut being considered is consistent. On any token, its vector $gcut$ is a consistent global state *iff* for all i, $depend[i] \leq gcut[i]$.
- $gstate$: Vector representation of global state corresponding to vector $gcut$. It is sufficient to keep only the states relevant to the predicate B.
- $eval$: Evaluation of B on $gstate$. The evaluation is either true or false; in our notation we use the values: $\{predtrue, predfalse\}$.
- $target$: A pointer to the unique event in the computation for which a token has to wait. The event need not belong to the local process.

A token waits at a slicer process P_i under three specific conditions:

(C1) The token is for process S_i and it has computed $J_B(pred(e))$, $pred(e)$ being the immediate predecessor event of e, and is waiting for the arrival of e.

(C2) The token is for process S_i and it is computing $J_B(f)$, where f is an event on P_i prior to e. The computation of $J_B(f)$ requires the token to advance along process P_i.

(C3) The token is for process S_j such that $j \neq i$, and it is computing $J_B(f)$ which requires the token to advance along process P_i.

On occurrence of each relevant event $e \in E_i$, the computation process P_i performs a *local* enqueue to *slicer* S_i, with the details of this event. Note that P_i and its slicer S_i are modeled as two threads on the same process, and therefore the *local* enqueue is simply an insertion into the queue (that is shared between the threads on the same process) of the *slicer*. The message contains the details of event e, i.e. the event identifier $<eid, pid>$, the corresponding vector clock $e.V$, and P_i's local state $localstate_e$ corresponding to e. The steps of the presented routines are explained below:

Algorithm 1: Algorithm at S_i

```
1  ReceiveEvent (Event $e$,State $localstate_e$)
2      save $<e.eid, localstate_e>$ in local state map $<procstates>$
3      foreach waiting token $t$ at $S_i$ do
4          if ($t.target = e$) then //$t$ waiting for event $e$
5              AddEventToToken ($t$,$e$)
6              ProcessToken ($t$)
7
8      end
9  AddEventToToken (Token $t$,Event $e$)
10     $t.gstate[e.pid] \leftarrow procState[e.eid]$
11     $t.gcut[e.pid] \leftarrow e.eid$
12     if ($t.pid = i$) then //my token: update token's $event$ pointer
13         $t.event = e$
14     end
15     $t.depend \leftarrow max(t.depend, e.V)$ // set causal dependency
16 ProcessToken (Token $t$)
17     if ($t.gcut$ is $inconsistent$) then
           /* find $k$ : $t.gcut[k] < t.depend[k]$ */
18         $t.target \leftarrow t.gcut[k] + 1$ // set desired event
19         send $t$ to $S_k$
20     else // $t.gcut$ is consistent
21         EvaluateToken ($t$)
22     end
23 EvaluateToken(Token $t$)
24     if $B(t.gstate)$ then //$B$ is true on cut given by $t.gcut$
25         $t.eval \leftarrow predtrue$
26         send $t$ to process $S_{t.pid}$
27     else // $B$ is false on $t.gstate$
28         $t.eval \leftarrow predfalse$
       /* $P_k$ : forbidden process in $t.gstate$ for $B$ */
29         $t.target \leftarrow t.gcut[k] + 1$
30         send $t$ to $S_k$
31     end
32 ReceiveToken (Token $t$)
33     if ($t.eval = predtrue$) $\wedge$ ($t.pid = i$) then //my token, $B$ true
34         output($t.pid, t.eid, t.gcut$)
       /* token waits for the next event */
35         $t.target \leftarrow t.gcut[i] + 1$
36         $t.waiting \leftarrow true$
37     else //either incosistent cut, or predicate false
38         $newid \leftarrow t.target$ // id of event $t$ requires
39         if ($\exists f \in localEvents : f.id = newid$) then
40             //required event has happened
41             AddEventToToken ($t$,$f$)
42             EvaluateToken ($t$)
43         end
44         //else, the token remains in $waiting$ state
45     end
46 ReceiveStopSignal
47     foreach token $t$ : $t.pid \neq i$ do
48         //not my token, send back to parent
49         send t to $S_{t.pid}$
50     end
```

ReceiveEvent (Lines 1-8): On receiving the details of event e from P_i, S_i adds them in the mapping of P_i's local states $procstates$ (line 2). It then iterates over all the waiting tokens, and checks their $target$. For each token that has e as the target (required event to make progress), S_i updates the state of the token, and then processes it.

`AddEventToToken` (Lines 9-15): To update the state of some token t on S_i, we advance the candidate cut to include the new event by setting $t.gcut[i]$ to the id of event e. If S_i is the parent process of the token (T_i), then the $t.event$ pointer is updated to indicate the event id for which token is computing the join-irreducible cut that satisfies the predicate. The causal-dependency is updated at line 15, which is required for checking whether or not the cut is consistent.

`ProcessToken` (Lines 16-22): To process any token, S_i first checks that the global state in the token is consistent (line 17) and at least beyond the global states that were earlier evaluated to be false. For t's evaluation of a global cut $t.gcut$ to be consistent, $t.gcut$ must be at least $t.depend$. This is verified by checking the component-wise values in both these vectors. If some index k is found where $t.depend > t.gcut$, the token's cut is inconsistent, and $t.gcut$ must be advanced by at least one event on P_k, by sending the token to *slicer* of P_k. If the cut is consistent, the predicate is evaluated on the variables stored as part of $t.gstate$ by calling the `EvaluateToken` routine.

`EvaluateToken` (Lines 23-31): The cut represented by $t.gstate$ is evaluated; if the predicate is true, then the token has computed $J_B(e)$ for the event $e = <t.pid, t.eid>$. The token is then sent to its parent *slicer*. If the evaluation of the predicate on the cut is false, the $target$ pointer is updated, at line 29, and the token is sent to the *forbidden* process on which the token must make progress.

`ReceiveToken` (Lines 32-45): On receiving a token, the *slicer* checks if the predicate evaluation on the token is true, and the token is owned by the *slicer*. In such a case, the *slicer* outputs the cut information, and now uses the token to find $J_B(succ(e))$, where $succ(e)$ denotes the event that locally succeeds e. This is done by setting the new event id in $t.target$ at line 35, and then setting the waiting flag (line 36). If the predicate evaluation on the token is false, then the $target$ pointer of the token points to the event required by the token to make progress. S_i looks for such an event (line 39), and if it has been reported to S_i by P_i, then adds that event (line 41) to the token and processes it (line 42). In case the desired event has not been reported yet to the *slicer* process, the token is retained at the process S_i and is kept in the waiting state until the required event arrives. Upon arrival of the required event, its details are added to the token and the token is processed.

Note: The notation of $target \leftarrow t.gcut[i] + 1$ means that if the $t.gcut[i]$ holds the event id $<pid, eid>$, then the $target$ pointer is set to $<pid, eid + 1>$.

`ReceiveStopSignal` (Lines 46-50): For finite computations, a single token based termination detection algorithm is used in tandem. Any one of the *slicer* process, let us assume S_1 holds a separate *stop* token. Whenever P_1 is finished with its computation, it sends a signal to S_1, and S_1 in turn checks if it has any tokens on which it has not updated the local events from P_1. Only after all such updates and processing is completed, and there are no more local events to process, S_1 forwards the *stop* token to S_2, and so on. When S_1 receives the *stop* token from S_n, it can deduce that all the *slicer* processes have completed processing all the events from their local queues, and there is no *slicing* token that can be advanced further. S_1 then sends the 'stop' signal to all the *slicer* processes, including itself. On receiving the 'stop' signal, S_i sends all the *slicing* tokens that do not belong to it back to their parent processes.

Note that the routines require atomic updates and reads on the local queues, as well as on tokens present at S_i. In the interest of space we skip presenting the lower level implementation details, that involve common local synchronization techniques.

A. Example of Algorithm Execution

This example illustrates the algorithm execution steps for one possible run (real time observations) of the computation shown in Fig. 2, with respect to the predicate B = "all channels empty". The algorithm starts with two slicing processes S_1 and S_2, each having a token – T_1 and T_2 respectively. The *target* pointer for each token T_i is initialized to the event $<i, 1>$. When event a is reported, S_1 adds its details to T_1, and on its evaluation finds the predicate "all channels empty" to be true, and outputs this information. It then updates $T_1.target$ pointer and waits for the next event to arrive. Similar steps are performed by S_2 on T_2 when e is reported.

When b is reported to S_1, and T_1 is evaluated with the updated information, the predicate is false on the state $[b]$. Given that b is a message send event, it is obvious that for the channel to be empty, the message receive event should also be incorporated. Thus, S_1 sends T_1 to S_2 after setting the target pointer to the first event on S_2. On receiving T_1, S_2 fetches the information of its first event (e) and updates T_1. The subsequent evaluation still leads to the predicate being false. Thus S_2 retains T_1 and waits for the next event.

When f is reported, S_2 updates both T_1 and T_2 with f's details. S_2's evaluation on $T_1.gstate$, represented by $[b, f]$ is true, and as per line 26, T_1 is sent back to S_1 where the consistent cut $[b, f]$ is output. T_1 now waits for the next event. However, after being updated with the details of event f, the resulting cut on T_2 is inconsistent, as the message-receive information is present but the information regarding the corresponding send event is missing. By using the vector clock values, T_2's target would be set to the id of message-send event b. S_2 would then send T_2 to S_1. On receiving T_2, S_1 finds the required event (looking at $T_2.target$) and after updating T_2 with its details, evaluates the token. The predicate is true on $T_2.gstate$ now, and T_2 is sent back to S_2. On receiving T_2, S_2 outputs the consistent cut $[b, f]$, and waits for the next event. On receiving details of event c, and adding them to the waiting token T_1, the predicate is found to be true again on T_1, and S_1 outputs $[c, f]$. Similarly on receiving g, S_2 performs similar steps and outputs $[b, g]$. Note that the consistent cuts $[a, b]$ and $[c, g]$, both of which satisfy the predicate are not enumerated as they are not join-irreducible, and can be constructed by the unions of $[a]$, $[b]$ and $[c, f]$, $[b, g]$ respectively.

B. Proof of Correctness

This section proves correctness and termination (for finite computations) of the distributed algorithm of Algorithm 1. The proofs presented here are for finite computations. The correctness argument can be easily extended to infinite computations.

Lemma 1. *The algorithm presented in Algorithm 1 does not deadlock.*

Proof: The algorithm involves n tokens, and none of the tokens wait for any other token to complete any task. With non-lossy channels, and no failing processes, the tokens are never lost. The progress of any token depends on the $target$ event, and as per lines 4-7, whenever an event is reported to a *slicer*, it always updates the tokens with their $target$ being this event. Thus, the algorithm can not lead to deadlocks. ■

Lemma 2. *If a token T_i is evaluating $J_B(e)$ for $e \in E_i$, assuming $J_B(e)$ exists, and if $T_i.gcut < J_B(e)$, then $T_i.gcut$ would be advanced in finite time.*

Proof: If during the computation of $J_B(e)$, at any instance $T_i.gcut < J_B(e)$, then there are two possibilities for $gcut$:
(a) $gcut$ is consistent: This means that the evaluation of predicate B on $gcut$ must be false, as by definition $J_B(e)$ is the least consistent cut that satisfies B and includes e. In this case, by line 29 and subsequent steps, the token would be forced to advance on some process.
(b) $gcut$ is inconsistent: The token is advanced on some process by execution of lines 17-19. ■

Lemma 3. *While evaluating $J_B(e)$ for event $e \in E_i$ on token T_i, if $T_i.gcut < J_B(e)$ currently and $J_B(e)$ exists then the algorithm eventually outputs $J_B(e)$.*

Proof: By Lemma 2, the global cut of T_i would be advanced in finite time. Given that $J_B(e)$ exists, we know that by the linearity property, there must exist a process on which T_i should progress its $gcut$ and $gstate$ vectors in order to reach the $J_B(e)$; lines 29-31 ensure that this forbidden process is found and T_i sent to this process. By the previous Lemma, the cut on the T_i would be advanced until it matches $J_B(e)$. By line 34 of the algorithm, whenever $J_B(e)$ is reached, it would be output. ■

Lemma 4. *For any token T_i, the algorithm never advances $T_i.gcut$ vector beyond $J_B(e)$ on any process, when searching $J_B(e)$ for $e \in E_i$.*

Proof: The search for $J_B(e)$ starts with either an empty global state vector, or from the global state that is at least $J_B(pred(e))$, where $pred(e)$ is the immediate predecessor event of e on S_i. Thus, till $J_B(e)$ is reached, the global cut under consideration is always less than $J_B(e)$. From the linearity property of advancing on the forbidden process, and Lemma 2, the cut would be advanced in finite time. Whenever the cut reaches $J_B(e)$, it would be output as per Lemma 3 and the token would be sent back to its parent slicer, to either begin the search for $succ(e)$ or to wait for $succ(e)$ to arrive ($succ(e)$ being the immediate successor of e). Thus, $T_i.gcut$ would never advance beyond $J_B(e)$ on any process when searching for $J_B(e)$ for any event e. ■

Lemma 5. *If token T_i is currently not at S_i, then T_i would return to S_i in finite time.*

Proof: Assume T_i is currently at S_j ($j \neq i$). S_j would advance $T_i.gcut$ in finite time as per Lemma 2. With no deadlocks (Lemma 1), and by the results of Lemma 3 and Theorem 4, we are guaranteed that if $J_B(T_i.event)$ exists then within a finite time, $T_i.gcut$ vector would be advanced to $J_B(T_i.event)$ and T_i would be sent back to S_i. If $J_B(T_i.event)$ does not exist then at least one slicer process S_k would run out of all its events while attempting to advance on $T_i.gcut$. In such a case, knowing that there are no more events to process, S_k would send T_i back to S_i (lines 46-50). ■

Theorem 2. (Termination)*: For a finite computation, the algorithm terminates in finite time.*

Proof: We first prove that for any event $e \in E_i$, computation of finding $J_B(e)$ with token T_i takes finite time. By Lemma 2, T_i always advances in finite time while computing $J_B(e)$. If $J_B(e)$ exists, then based on this observation within a finite time the token T_i would advance its $gcut$ to $J_B(e)$, if it exists. By Lemma 3, the algorithm would output this cut, thus finishing the $J_B(e)$ search and as per Theorem 4 would not advance any further for $J_B(e)$ computation. Thus, if $J_B(e)$ exists then it would be output in finite time. By Lemma 5 the token would be returned to its parent process and the $J_B(e)$ computation for $e \in E_i$ would finish in finite time.

If $J_B(e)$ does not exist, then as we argued in Lemma 5 some *slicer* would run out of events to process in the finite computation, and thus return the token to S_i, which would result in search for $J_B(e)$ computation to terminate. As each of these steps is also guaranteed to finish in finite time as per above Lemmas, we conclude that $J_B(e)$ computation for $e \in E_i$ finished in finite time.

Now we can apply this result to all the events in E, and guarantee termination in finite time. ■

Theorem 3. *The algorithm outputs all the elements of J_B.*

Proof: Whenever any event $e \in E$ occurs, it is reported by some process P_i on which it occurs, to the corresponding slicer process S_i. Thus e can be represented as $e \in E_i$. If at the time e is reported to S_i, T_i is held by S_i then by Lemmas 2 and 3, it is guaranteed that the algorithm would output $J_B(e)$. If S_i does not hold the token T_i when e is reported to it, then by Lemma 5, T_i would arrive on S_i within finite time. If S_i has any other events in its processing queue before e, then as per Theorem 2, S_i would finish those computations in finite time too. Thus, within a finite time, the computation for finding $J_B(e)$ with T_i would eventually be started by S_i. Once this computation is started, the results of Lemmas 2 and 3 can be applied again to guarantee that the algorithm would output $J_B(e)$, if it exists.

Repeatedly applying this result to all the events in E, we are guaranteed that the algorithm would output $J_B(e)$ for every event $e \in E$. Thus the algorithm outputs all the join-irreducible elements of the computation, which by definition together form J_B. ∎

Theorem 4. *The algorithm only outputs join-irreducible global states that satisfy predicate B.*

Proof: By Lemma 4, while performing computations for $e \in E_i$ on token T_i, the algorithm would not advance on token T_i beyond $J_B(e)$. Since only token T_i is responsible for computing $J_B(e)$ for all the events $e \in E_i$, the algorithm would not advance beyond $J_B(e)$ on any token. In order to output a global state that is not join-irreducible we must advance the cut of at least one token beyond a least global state that satisfies B. The result follows from the above assertions. ∎

Theorem 2 guarantees termination, and correctness follows from Theorems 3, and 4.

IV. Optimizations

The distributed algorithm presented in the previous section is not optimized to avoid redundant token messages, as well as duplicate computations. Whenever a *slicer* process S_i needs to send any token to another process S_k, it should first check if it currently holds the token T_k, and if the desired information is present in T_k. If the information is available, the token T_i can be updated with the information without being sent to S_k; and only if the details of required event are not available locally, the token is sent to S_k. These steps are captured in the procedure `SendIfNeeded` shown in Algorithm 2.

There are additional optimizations that significantly reduce the number of token messages. It is easy to observe that in the proposed form of Algorithm 1, the algorithm performs many redundant computations. This redundancy is caused by computations of $J_B(e)$ and $J_B(f)$ where $e \neq f$, and $J_B(e) = J_B(f)$. In this case, given that both the join-irreducible consistent cuts are same, it would suffice that the algorithm only compute either of them. For this purpose, we first present some additional results:

Lemma 6. $f \in J_B(e) \Rightarrow J_B(f) \subseteq J_B(e)$.

Proof: $J_B(f)$ is the least consistent cut of the computation that satisfies the predicate, and contains f. $J_B(e)$ includes f, and satisfies the predicate. Therefore $J_B(f) \subseteq J_B(e)$. ∎

Lemma 7. $f \in J_B(e) \wedge e \in J_B(f) \Rightarrow J_B(e) = J_B(f)$.

Proof: Apply previous Lemma twice. ∎

Algorithm 2: SendIfNeeded at S_i

```
SendIfNeeded (Token $t$, int $k$)
/* $k$: id of the slicer process to which $t$ should be sent */
while $(k \neq i) \wedge$ (have $token_k$) do
    /* $t$ should be sent to $S_k$, and $S_i$ has $S_k$'s token */
    if ($t.target = token_k.event$) then //$token_k$ has info of $t$'s
    //target event
        $t.gcut[k] \leftarrow token_k.gcut[k]$
        $t.depend[k] \leftarrow token_k.depend[k]$
        $t.state[k] \leftarrow token_k.state[k]$
        if ($t.gcut$ is inconsistent) then //still inconsistent
            /* find $j : t.gcut[j] < t.depend[j]$ */
                $t.target \leftarrow t.gcut[j] + 1$
                $k \leftarrow j$ //set $k$ for while condition
        else // $t.gcut$ is consistent now, evaluate
            EvaluateToken ($t$)
        end
    else // desired event details not in $token_k$
        $break$
    end
end
/* desired token or event info not present */
if ($t.target.pid \neq i$) then
    /* target event on some other process */
    send $t$ to $S_k$
end
```

Lemma 8. $e \rightarrow f \wedge f \in J_B(e) \Rightarrow J_B(f) = J_B(e)$.

Proof: By Lemma 6, $f \in J_B(e)$ implies that $J_B(f) \subseteq J_B(e)$ must hold. Given $e \rightarrow f$, by the consistency requirement $J_B(f)$ must contain e. Thus, $J_B(e) \subseteq J_B(f)$. ∎

In order to prevent computations that result in identical join-irreducible states, we modify the proposed distributed algorithm of Algorithm 1 to incorporate Lemmas 7, and 8. The modified algorithm is presented in Algorithm 3.

We do not reproduce the functions `ReceiveEvent`, `EvaluateToken`, `ReceiveStopSignal` and `SendIfNeeded` in the modified algorithm (in Algorithm 3), as they remain identical to their earlier versions. In the optimized algorithm an additional variable, $currentE$ - at each *slicer* process S_i, is used as a local pointer to keep track of the event e for which S_i is currently computing $J_B(e)$. Tokens also stores this information (with $token.event$), however the token T_i is not always present on S_i. By keeping $currentE$ updated, even in absence of token T_i, the *slicer* process S_i can delay the progress of other tokens whenever it suspects that these tokens may undergo the same $J_B(e)$ computation that is being considered by T_i. For stopping possibly duplicate computations, a flag, called $stalled$, is maintained in each token. By setting the $stalled$ flag on any token, a slicer removes the token from the set of $waiting$ tokens; and no updates are performed on tokens that are in the $stalled$ state. These modifications allow *slicer* processes to delay the computation progress on $stalled$ tokens, and ensure that no two tokens finish and output any two join-irreducible consistent cuts that are equal. The optimized algorithm also makes use of the type information of events, for identifying if an event is a send of a message (denoted by type

Algorithm 3: Optimized Routines at S_i

```
1  AddEventToToken (Token $t$,Event $e$)
2      $t.gstate[e.pid] \leftarrow procState[e.eid]$
3      $t.gcut[e.pid] \leftarrow e.eid$
4      if ($t.pid = i$) then //my token: update current event
5          $currentE = e.eid$
6          $t.event = e$
7      end
8      $t.depend \leftarrow max(t.depend, e.V)$
9  ProcessToken (Token $t$)
10     if ($t.gcut$ is $inconsistent$) then
11         if ($t.event.type = MSGRECV$) then // message receive
           event
12             $t.target \leftarrow t.event.sender$
13             if ($t.pid = i$) then // my token
14                 $t.stalled \leftarrow true$ //stall, wait to hear from target
15             end
16             else // progress other processes' token
                   /* find $k$ s.t. $t.gcut[k] < t.depend[k]$ */
17                 $t.target \leftarrow t.depend[k]$
18                 SendIfNeeded ($t$) // use optimized send approach
19             end
20         end
21     end
22     else // $t.gcut$ is consistent, evaluate its state
23         EvaluateToken ($t$)
24     end
25 ReceiveToken (Token $t$)
26     if ($t.eval = predtrue$) $\wedge$ ($t.pid = i$) then // my token, $B$ true
27         output($t.pid, t.eid, t.gcut$)
28         UpdateStalledTokens ($t.eid$,$t.gcut$)
           /* token waits for the next event */
29         $t.target \leftarrow t.gcut[i] + 1$
30         $t.waiting \leftarrow true$
31     end
32     else //either incosistent cut, or predicate false
33         $newid \leftarrow t.target$        // id of $t$'s required event
34         if $t.event \not\rightarrow currentE \wedge newid \geq currentE \wedge i > t.pid$
           then // no causal-dependency, and symmetry breaking
35             $t.stalled \leftarrow true$        //stall the token
36         end
37         else if ($\exists f \in localEvents : f.id = newid$) then
38             // required event has happened
39             AddEventToToken ($t$,$f$)
40             EvaluateToken ($t$)
41         end
42     end
43 ReceiveCut (Event $event$,Vector $cutV$,Vector $stateV$)
44     foreach $stalled$ $token$ $t$ $at$ $S_i$ do // check if $t$ can be updated
45         CopyCutIfNeeded ($t, event, cutV, stateV, i$)
46     end
```

Algorithm 4: Helper Routines at S_i

```
1  UpdateStalledTokens (Token $t'$)
2      foreach $stalled$ $token$ $t$ $at$ $S_i$ do
3          CopyCutIfNeeded ($t, t'.event, t'.cutV, t'.stateV, t.pid$)
4      end
5      if ($t'.event.type = MSGSEND$) then
           /* send cut details to message recipient process; $k$: id of the
           message recipient process */
6          $S_k$.ReceiveCut ($t'.event, t'.gcut, t'.gstate$)
7      end
8  CopyCutIfNeeded (Token $t$, Event $event$, Vector $cutV$, Vector
   $stateV$, int $ignore$)
9      if $t.target = event$ then // $t$ was waiting for update from $event$
10         foreach $j$ $in$ 1 $to$ $n$ $s.t.$ $j \neq ignore$ do   //copy relevant details
11             $t.gcut[j] \leftarrow cutV[j]$
12             $t.depend[j] \leftarrow cutV[j]$
13             $t.gstate[j] \leftarrow stateV[j]$
14         end
           /* clear stalled state; ensure no duplicate output */
15         $t.stalled \leftarrow false$
16         $t.eval \leftarrow predfalse$
17         if ($t.event \in cutV$) then
               /* cuts are same, move on to next event */
18             $t.target \leftarrow t.gcut[i] + 1$
19             if ($t.pid \neq i$) then
20                 send $t$ to $S_{t.pid}$
21             end
22         end
23         else
               /* cuts not same, resume token's computation */
24             ProcessToken ($t$)
25         end
26     end
```

$MSGSEND$) or a receive of a message (denoted by type $MSGRECV$). The modifications are briefly explained below:

Algorithm 3: Optimized Routines

Lines 11-15 and 43-46: If the type of an event $e \in E_i$ indicates that the event is a message receipt, then the algorithm stalls the token T_i, and sets its 'target' as the message send event, f. This step is to incorporate Lemma 8 in speculative manner. Whenever the corresponding *slicer* process of message send event (f) finishes computing the $J_B(f)$, it informs S_i about the computed cut. S_i on receiving this cut, calls the helper sub-routine CopyCutIfNeeded (shown in Algorithm 4) that checks if e belongs to $J_B(f)$ and thus $J_B(e)$ computation is not needed; otherwise S_i restarts the computation for $J_B(e)$.

Lines 34-36: These steps incorporate Lemma 7 in speculative manner. The first condition $t.event \not\rightarrow currentE$ ensures that if the current computation on token T_i is causally dependent on the computation on token t, then t is not stalled. The second condition is evaluated only if $t.event$ and $currentE$ are not causally related, i.e. they are concurrent. If this is the case, then the check ($newid \geq currentE \wedge i > t.pid$) ensures that the computation of $J_B(t.event)$ does not progress beyond the current ongoing computation on S_i, and performs symmetry breaking (so that there is no deadlock) in favor of the token/process with larger process id. This guarantees that whenever the cuts of two concurrent events are same, only one of the tokens (with the smaller process id) finishes computing the cut, and thus duplicate computations are not performed.

Line 28: Whenever S_i finishes computing the $J_B(e)$ for the event $e \in E_i$, it tries to update all the tokens that were speculatively stalled to avoid computing the same cut. The steps involved are explained next.

Algorithm 4: Helper Routines

Lines 1-7: For each token that is stalled, either locally or at some other *slicer* process, due to the event $t'.event$, update (tokens present locally) or notify (at some other process) it. The notification to other *slicer* processes is performed by sending the details of the cut to them. If the stalled tokens infer, using the checks of lines 9 and 17, that their cuts (if computed) and $t'.gcut$ would be same by the application of Lemmas 8 and 7, then they copy the cut details of t' and move on to the next events on their respective processes.

A. Example of Optimized Algorithm Execution

We revisit the example presented in section III-A for the distributed algorithm run, in order to show the difference in execution for the optimized algorithm. When f is reported to S_2, the earlier version of the algorithm has to update the token T_2, and send it to S_1 in order to make the cut on T_2 consistent. The optimized algorithm uses the checks on lines 11-15 in Algorithm 3, and determines that f being a message-receive event, the $J_B(f)$ computation should not be started until the corresponding message-send event's computation is reported to S_2. Thus T_2 would be kept in $stalled$ state, until T_1 finishes the $J_B(b)$ computation. When $J_B(b)$ computation on T_1 is finished, with $J_B(b) = [b, f]$, then as per line 28 of Algorithm 3, S_1 would send the information of b's join-irreducible cut to S_2. On receiving the cut details, S_2 would try to update its stalled token T_2, and as per lines 9-18, it would infer that $J_B(f) = J_B(b)$. Thus, it would just copy the details of the cut as the result for $J_B(f)$, and move on to computing $J_B(g)$.

B. Proof of Correctness

To prove the correctness of the optimized version of the algorithm, it suffices to show that this version does not introduce deadlocks, and as desired – does not enumerate duplicate join-irreducible consistent cuts.

Theorem 5. *Algorithm 3 cannot lead to deadlocks.*

Proof: Let us assume that the algorithm leads to a deadlock. Thus, even in presence of events required to process, a set of tokens is not able to make progress, with every token in this set being in the stalled state. There are two possible scenarios for this:
(a): The tokens are stalled such that for each token T_i, the event $T_i.event$, for which it is computing the join-irreducible cut, is concurrent to every other token T_j's event $T_j.event$. As all the tokens have unique positive ids, there is a unique minimum id. By the required check performed at line 34 in Algorithm 3, such a token can never be stalled. Thus, we have a contradiction.
(b): All the tokens are stalled such that for at least one pair of tokens T_i and T_j, $T_i.event \rightarrow T_j.event$. In such a case, by the check performed at line 11 in Algorithm 3, T_i's token can never be stalled at S_j. Thus T_i must be stalled at some other process S_k such that $T_i.event$ is concurrent with $T_k.event$. But either $i < k$ or $i > k$, and in either case, by applying the result of case (a) above, we are guaranteed that T_i would eventually make progress. Thus T_i would not remain stalled forever. Again, we have a contradiction. ■

Theorem 6. *If $J_B(e) = J_B(f)$ for two distinct events e and f, then only one of these cuts is enumerated in Algorithm 3.*

Proof: Given that $e \neq f$, then either: (a) $e||f$ (both are concurrent), or $e \rightarrow f$ (assume without loss of generality). If $e||f$, then as per line 34 one of their tokens would be stalled on the other process. Again, without loss of generality assume f's token was stalled on e's process. Thus, given that there can be no deadlocks, the $J_B(e)$ computation would eventually finish, and update f's token to indicate that it should move to the successor of f (as per the steps in Algorithm 4). Hence, $J_B(f)$ would not be enumerated.
If $e \rightarrow f$, then f's token would be stalled on its own process, until the completion of $J_B(e)$ computation, upon which f's token would be made to move to the event that is the successor of f. Again $J_B(f)$ would not be enumerated. ■

C. Analysis

Each token T_i processes every event $e \in E_i$ once for computing its $J_B(e)$. If there are $|E|$ events in the system, then in the worst case T_i does $O(n|E|)$ work, because it takes $O(n)$ to process one event. We are assuming here that evaluation of B takes $O(n)$ time given a global state. There are n tokens in the system, hence the total work performed is $O(n^2|E|)$. Since there are n slicing processes and n tokens, the average work performed is $O(n|E|)$ per process. In comparison, the centralized algorithm (either online or offline) requires the *slicer* process to perform $O(n^2|E|)$ work.

Let $|S|$ be the maximum number of bits required to represent a local state of a process. The actual value of $|S|$ is subject to the predicate under consideration, as the resulting number/type of the variables to capture the necessary information for predicate detection depends on the predicate. The centralized online algorithm requires $O(|E||S|)$ space in the worst case; however it is important to notice that all of this space is required on a single (central slicer) process. For a large computation, this space requirement can be limiting. The distributed algorithm proposed above only consumes $O(|E_i||S|)$ space per slicer. Thus, we have a reduction of $O(n)$ in per slicer space consumption.

The token can move at most once per event. Hence, in the worst case the message complexity is $O(|E|)$ per token. Therefore, the message complexity of the distributed algorithm presented here is $O(n|E|)$ total for all tokens. The message complexity of the centralized online slicing algorithm is $O(|E|)$ because all the event details are sent to one (central) slicing process. However, for conjunctive predicates, it can be observed that the message complexity of the optimized version of the distributed algorithm is also $O(|E|)$. With speculative stalling of tokens, only unique join-irreducible cuts are computed. This means that for conjunctive predicates, a token only leaves (and returns to) S_i, $O(|E_i|)$ times. As there are n tokens, the overall message complexity of the optimized version for conjunctive predicates is $O(|E|)$.

V. Evaluation & Discussion

As indicated by the analysis of the distributed algorithm in Section IV-C, the distributed algorithm reduces the worst case complexity for the total work per slicer process by an order of magnitude. To better understand the actual gains of the distributed algorithm, we implement both centralized and distributed algorithms, and evaluate their performance based on the experimental results for slicing on the same set of computations.

The experiments were performed on a 64 bit, 8 processor (2.3 GHz) machine with 4GB memory, running Linux 3.2.0-32 kernel. Each process is allowed to run till it executes its local program counter to a fixed upper-bound. For the reported results the local program counter upper-bound was set to 100. Message activity was decided in a randomized manner. After each local state change the processes could send a message, with a probability of 0.8, to a randomly chosen process. The number of processes in the computation was varied from 2 to 10. We monitor the total object sizes of the centralized slicer, and each of the distributed slicer processes while they find the slice for an ongoing computation. We also monitor the total number of messages received by all the slicer processes during the online computations. For any distributed slicer S_i, the reported number of messages includes the token messages received by it from other slicers. We present a comparison of the centralized and distributed algorithms in terms of maximum values of both the consumed space and total messages per slicer process.

Fig. 3(a) plots the ratio of maximum space consumed by any distributed slicer object and the maximum space used by the centralized slicer (for the same computation), against the number of processes in the computation. The space consumption is evaluated at pre-determined check points that are symmetric for both centralized and distributed algorithms. Fig. 3(b) presents the maximum number of messages received by centralized slicer and that received by any distributed slicer for the same instances of computations.

VI. Related Work

The distributed algorithm presented in this paper constructs the slice of a distributed computation with respect to a regular state based predicate. The constructed slice can then be used to determine if some consistent cut of the computation satisfies the predicate. This is referred to as the problem of detecting a predicate under *possibly* modality [1]. In [1], a predicate is detected by exploring the complete lattice of consistent cuts in a breadth first manner. Alagar et. al. [6] use a depth first traversal of the computation lattice to reduce space complexity. The algorithms in [1] and [6] can handle arbitrary predicates, but in general have exponential time complexity. In contrast, the slicing algorithm presented in this paper for a *regular* predicate has polynomial time complexity.

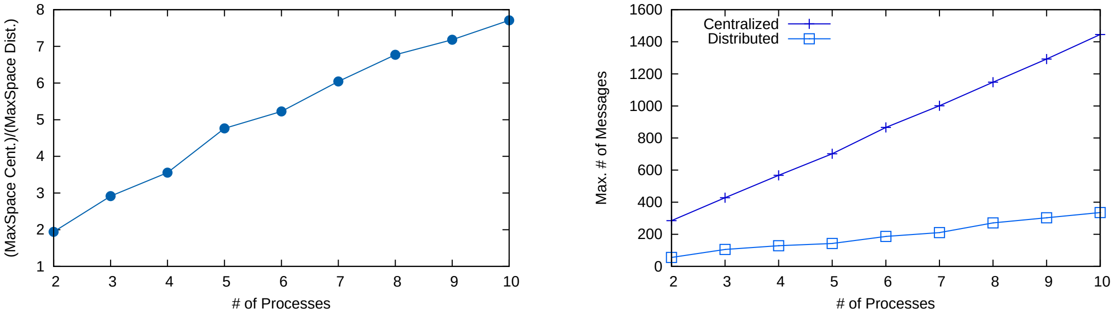
In this paper we assume a static distributed system. Predicate detection algorithms have been proposed for dynamic systems (e.g. [18], [19], [20], [21], [22]), where processes may leave or join. However, these algorithms detect restricted classes of predicates like stable predicates and conjunctive local predicates, which are less general than regular predicates. In computation slicing, we analyze a single *trace* (or execution) of a distributed program for any violation of the program's specification. Model checking (cf. [23]) is a formal verification technique that involves determining if (all traces of) a program meets its specification. Model checking algorithms conduct reachability analysis on the state space graph, and have a time complexity that is exponential in number of processes.

Partial order methods (cf. [24]) aim to alleviate the state-explosion problem by minimizing the state space for predicate detection. This is done by exploring only a subset of the interleavings of concurrent events in a computation, instead of all possible interleavings. However, predicate detection algorithms based on partial order methods still have exponential time complexity, in the worst case. In this paper, the focus is on generating the slice with respect to a predicate. Partial order methods such as [25] can be used in conjunction with slicing to explore the state space of a slice in a more efficient manner [26].

The work presented in this paper is related to runtime verification (cf. [27]), which involves analyzing a run of a program to detect violations of a given correctness property. The input program is instrumented and the trace resulting from its execution is examined by a monitor that verifies its correctness. Some examples of runtime verification tools are Temporal Rover [28], Java-MaC [29], JPaX [30], JMPaX [31], and jPredictor [32]. The Temporal Rover, Java-Mac and JPaX tools model the execution trace as a total order of of events, which is then examined for violations. In the JMPaX and jPredictor tools, as in our algorithm, the trace is modeled as a partial order of events. A lattice of consistent cuts of the computation is then generated, which is searched by the monitor. Further, these tools generate states not observed in the current trace, to predict errors that may occur in other runs, thereby increasing the size of the computation lattice. Chen et al. [33] note that computation slicing can be used to make tools like jPredictor more efficient by removing redundant states from the lattice. All of these tools are centralized in nature, where the events are collected at a central monitoring process. Sen et al. [34] present a decentralized algorithm that monitors a program's execution, but can only detect a subset of safety properties. The distributed algorithm presented by Bauer et al. [35] can handle a wider class of predicates, but requires the underlying system to be synchronous.

VII. Conclusion

In this paper, we presented a distributed online algorithm for performing *computation slicing*, a technique to abstract the computation with respect to a regular predicate. The resulting abstraction (*slice*) is usually much smaller, sometimes exponentially, in size. For regular predicates, by detecting the predicate only on the abstracted computation, one is guaranteed to detect the predicate in the full computation, which leads to an efficient detection mechanism. By distributing the task of abstraction among all the processes, our distributed algorithm reduces the space required, as well as computational load on a single process by a factor of $O(n)$. We also presented an optimized version of the distributed algorithm that does



(a) Comparison of Maximum Space per Slicer

(b) Max. messages per Slicer

Fig. 3: Comparison of Memory Usage and Total Messages for Conjunctive Predicate

not perform redundant computations, and requires reduced number of messages. The results of experimental evaluation (available in extended version of this paper at [11]) compare the performance of our distributed algorithm with that of the existing centralized algorithm.

References

[1] R. Cooper and K. Marzullo, “Consistent Detection of Global Predicates,” in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, USA, 1991, pp. 163–173.

[2] “Tilera - TilePro Processor,” http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf.

[3] R. Rotta, T. Prescher, J. Traue, and J. Nolte, “In-memory communication mechanisms for many-cores–experiences with the intel scc,” in *TACC-Intel Highly Parallel Computing Symposium (TI-HPCS)*, 2012.

[4] D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper, “High-performance rma-based broadcast on the intel scc,” in *SPAA*, 2012, pp. 121–130.

[5] N. Mittal, A. Sen, and V. K. Garg, “Solving Computation Slicing using Predicate Detection,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 18, no. 12, pp. 1700–1713, Dec. 2007.

[6] S. Alagar and S. Venkatesan, “Techniques to Tackle State Explosion in Global Predicate Detection,” *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 8, pp. 704–714, Aug. 2001.

[7] V. K. Garg, *Elements of Distributed Computing*. New York, NY: John Wiley and Sons, Incorporated, 2002.

[8] N. Mittal and V. K. Garg, “Techniques and Applications of Computation Slicing,” *Distributed Computing (DC)*, vol. 17, no. 3, pp. 251–277, Mar. 2005.

[9] A. Sen and V. K. Garg, “Formal Verification of Simulation Traces Using Computation Slicing,” *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 511–527, Apr. 2007.

[10] V. Ogale and V. K. Garg, “Detecting Temporal Logic Predicates on Distributed Computations,” in *Proceedings of the Symposium on Distributed Computing (DISC)*, 2007, pp. 420–434.

[11] H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal, “Distributed abstraction algorithm for online predicate detection,” *CoRR*, vol. abs/1304.4326, Apr 2013.

[12] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM (CACM)*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[13] F. Mattern, “Virtual Time and Global States of Distributed Systems,” in *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, 1989, pp. 215–226.

[14] C. J. Fidge, “Timestamps in Message-Passing Systems that Preserve the Partial-Ordering,” in *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, K. Raymond, Ed., Feb. 1988, pp. 56–66.

[15] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*. Cambridge, UK: Cambridge University Press, 1990.

[16] K. M. Chandy and L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems,” *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[17] V. K. Garg and N. Mittal, “On Slicing a Distributed Computation,” in *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, Apr. 2001, pp. 322–329.

[18] X. Wang, J. Mayo, and G. Hembroff, “Detection of a Weak Conjunction of Unstable Predicates in Dynamic Systems,” in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, 2010, pp. 338–346.

[19] D. Darling, J. Mayo, and X. Wang, “Stable Predicate Detection in Dynamic Systems,” in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, J. Anderson, G. Prencipe, and R. Wattenhofer, Eds. Springer Berlin Heidelberg, 2006, vol. 3974, pp. 161–175.

[20] D. Dhamdhere, S. R. Iyer, and E. K. K. Reddy, “Distributed Termination Detection for Dynamic Systems,” *Parallel Computing*, vol. 22, no. 14, pp. 2025 – 2045, 1997.

[21] P. Johnson and N. Mittal, “A Distributed Termination Detection Algorithm for Dynamic Asynchronous Systems,” in *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Montreal, Quebec, Canada, 2009, pp. 343–351.

[22] X. Wang, J. Mayo, G. Hembroff, and C. Gao, “Detection of Conjunctive Stable Predicates in Dynamic Systems,” in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, 2009, pp. 828–835.

[23] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 2000.

[24] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1996, vol. 1032.

[25] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu, “Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods,” in *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)*, Jul. 2000, pp. 264–279.

[26] N. Mittal and V. K. Garg, “Software Fault Tolerance of Distributed Programs using Computation Slicing,” in *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, Providence, Rhode Island, May 2003, pp. 105–113.

[27] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009.

[28] D. Drusinsky, “The Temporal Rover and the ATG Rover,” in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK, UK: Springer-Verlag, 2000, pp. 323–330.

[29] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, “Java-MaC: A Run-time Assurance Tool for Java Programs,” *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 218 – 235, 2001.

[30] K. Havelund and G. Rosu, “Monitoring java programs with java pathexplorer,” *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 200 – 217, 2001.

[31] K. Sen, G. Rosu, and G. Agha, “Detecting errors in multithreaded programs by generalized predictive analysis of executions,” in *Formal Methods for Open Object-Based Distributed Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3535, pp. 7–14.

[32] F. Chen, T. F. Şerbănuţă, and G. Roşu, “jPredictor: A predictive runtime analysis tool for Java,” in *ICSE ’08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 221–230.

[33] F. Chen and G. Roşu, “Parametric and sliced causality,” in *Proceedings of the 19th international conference on Computer aided verification*, ser. CAV’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 240–253.

[34] K. Sen, A. Vardhan, G. Agha, and G. Rosu, “Efficient decentralized monitoring of safety in distributed systems,” in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 418–427.

[35] A. Bauer and Y. Falcone, “Decentralised ltl monitoring,” in *FM 2012: Formal Methods*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Mry, Eds. Springer Berlin Heidelberg, 2012, vol. 7436, pp. 85–100.